Splice: An Automated Framework for Cost- and Performance-Aware Blending of Cloud Services

Myungjun Son*, Shruti Mohanty*, Jashwant Raj Gunasekaran†, Aman Jain‡,
Mahmut Taylan Kandemir*, George Kesidis*, Bhuvan Urgaonkar*
* The Pennsylvania State University, USA, † Adobe Research, USA, ‡ Microsoft Corporation, USA
{mjson, sxm1743, mtk2, gik2, buu1}@psu.edu, jgunasekaran@adobe.com, Aman.Jain@microsoft.com

2022 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-6654-9956-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCGRID54584.2022.00021

***Abstract*—With the rapid growth of users adopting public clouds to run their applications, the types of resources procured from the different public cloud resource offerings are critical in simultaneously achieving satisfactory performance and reducing deployment costs. Typically, no one resource type can meet all application requirements, and thus combining different resource offerings is known to considerably reduce the performance-cost problem. However, it is non-trivial to use blended resources, due to the manual overhead of designing and implementing such blended approaches. Specifically, it necessitates rewriting the application code to suit a given resource and scaling it on demand.**

In order to overcome this manual hurdle, we take the first step by proposing Splice, an automated framework for cost- and performance-aware blending of IaaS and FaaS services. The three major goals of Splice are: (1) while cost-saving opportunities exist from blending resources, we aim to largely automate the blending process for public cloud services through a compiler-driven approach; (2) more specifically, we focus on automated blending of VMs and serverless functions; and (3) for serverless applications which contain multiple chained functions, we unearth the potential choices in determining a portion of the services to be blended cost-efficiently. We implement Splice on Amazon Web Services (AWS) using an Abstract Syntax Tree (AST), and extensively evaluate its effectiveness using several applications with real-world traces. Our experiments demonstrate that, through automated blending, Splice is able to reduce SLO violations by 31% compared to VM-based resource procurement schemes, while simultaneously minimizing costs by up to 32%.

***Index Terms*—automation, compiler, serverless, blending**

I. Introduction

The public cloud is being extensively used by different types of tenants for hosting their applications. Many such applications have performance requirements (e.g., latency), also called service-level objectives (SLOs). The types of cloud offerings procured to host these applications play a pivotal role in meeting their SLOs as well as determining the cost they incur. Henceforth, we refer to this as the *performance-cost problem.*

Public cloud resources have typically been procured via Infrastructure as a Service (IaaS) offerings such as virtual machines (VMs or instances), containers, block storage devices, etc. Offerings based on Platform/Function as a Service (PaaS/FaaS) and Software as a Service (SaaS) have also begun to be offered. For instance, a Machine Learning (ML) inference-based application can be hosted using a fully-managed solution, such as AWS SageMaker [7], or a semi-managed solution, such as AWS Elastic Container Service [5]. Key selling points for FaaS and SaaS offerings over IaaS include their finer-grained billing; reduced management and administrative effort/costs; and (alleged) reduced effort and cost of application development if such development is being done from scratch. From extant literature, e.g., [2], [33], it is evident that no single resource offering can best cater to all application requirements and blending different resource offerings greatly alleviates the performance-cost problem. Several recent works have focused on blending different types of IaaS offerings. Some recent works have also proposed blending IaaS with FaaS [20], [22], [40]. These blended solutions have been found to be more cost-efficient when compared to solutions based on a single offering type.

A major impediment in offering blended services is the manual effort required from developers to rewrite the application towards catering to each offering type that is being used in the blended solution. In fact, the human cost of these transformations itself constitutes a key constraint which has been ignored in the blending-related works cited above. This motivates the need to automate different aspects of such transformations by providing an efficient means to blend different resource offerings. Within the large space of blending possibilities, in this work, we focus on the following scenarios to address the performance-cost problem:

- **IaaS to IaaS+FaaS:** Code being migrated from a private cloud (e.g., an enterprise setting) to the public cloud "as is" (sometimes labeled "lift and shift") can benefit from refactoring parts of it to use FaaS offerings. Specifically, FaaS can be used as an agile transition mechanism when spinning up new VMs (note that serverless functions can be launched in a few milliseconds while VMs may take tens of seconds to minutes to start up), thus reducing over-provisioning of VMs to address the performance-cost problem.
- **FaaS to IaaS+FaaS:** Prior related research [20], [22] indicate that using FaaS in a standalone manner may be more expensive than using VMs depending on the workload properties, owing to the higher per-unit resource cost of FaaS. For this reason, cost savings may be possible for certain existing FaaS application chains by splitting them carefully and hosting suitable portions (especially stateful ones) within VMs.

While the above-mentioned blending scenarios can help

978-1-6654-9956-9/22/$31.00 ©2022 IEEE
DOI 10.1109/CCGrid54584.2022.00021

alleviate the performance-cost problem, as previously stated, the major downside in designing blended solutions is the manual overhead of rewriting applications to suit the diverse platforms in consideration. For instance, monolithic applications may need to be redesigned as multiple independent functions to leverage FaaS offerings. Moreover, for existing FaaS chains (multi-functions), deciding which functions to offload to VMs at runtime impacts both performance and cost. Therefore, it would be highly desirable to largely automate the identification and realization of such cost savings opportunities through blending. We take the first step towards this by proposing **Splice**, an automated framework for cost- and performance-aware blending of IaaS and FaaS services. The key components of Splice include the following: (1) an annotation schema in which users can specify the type of code transformations to be undertaken; (2) a compilation framework to automatically generate the code for different blended versions; and (3) a dynamic load balancer which can seamlessly switch between the blended versions based on user-specified cost and performance constraints. To this end, the key contributions of the paper are summarized below:

- For blending resources, we characterize the manual overhead involved in intermixing IaaS and FaaS, and provide key insights which enable efficient automation of the blending process.
- We build Splice, which consists of an annotation scheme, static compiler, controller, load-balancer and scaling policy, and can support both single function and multi-function applications.
- We implement Splice using various AWS cloud computing services and abstract syntax tree module. Splice is evaluated with real-world request arrival traces (WITS and Wikipedia) and benchmark suites (ResNet inference service, feature generation, matrix multiplication, and image processing) under different SLOs.
- We demonstrate that Splice minimizes SLO violations of inference queries by 16%-31% compared to employing only VMs. In addition, Splice saves up to 32% on costs when compared to standard resource procurement methods.
- We further analyze offloading different combinations of functions for a multi-function image processing application, and show that Splice reduces the SLO violations by up to 13% compared to using only VMs.

The remainder of this paper is structured as follows. Sec. II presents the necessary background and motivation for Splice. In Sec. III-A, we explain how to model the automation in Splice, followed by the design and implementation of Splice in Sec. III and IV. The experimental results are discussed in Sec. V, followed by related work in Sec. VI.

II. Background and Motivation

To appreciate why service blending may be desirable (over using only a single service type), consider Table I, which offers a glimpse of the trade-offs across different cloud service types. Recent literature [20], [22], [40] has explored these trade-offs and established them concretely. Specifically, these works have let both stateless and stateful tasks run on AWS Lambdas [25] (a FaaS/PaaS offering), as well as VMs (IaaS); AWS Lambda is generally a Function as a Service (FaaS) that can also manage automated scaling depending on user specifications (PaaS), and thus it is termed a "FaaS/PaaS offering." We explain the different blended versions that we consider in this work in detail.

A. IaaS to IaaS + FaaS

There are two primary reasons why using a PaaS offering may assist in lowering costs compared to IaaS, and they have been examined to different extents in related work (see Sec. VI for details). All of these apply to latency-critical workloads.

Going forward, we use the term "Lambdas" for PaaS and "VMs" for IaaS, while acknowledging that PaaS and IaaS are more general terms.

1) *Lambdas for handling fine time-scale variability:* For latency-critical workloads, auto-scaling techniques need to provision for close-to-peak needs. Lambdas may allow one to provision VMs for less than the peak. Lambdas also provide finer-grained pricing than containers inside VMs for highly intermittent applications. This benefit applies most readily to stateless workloads. Moreover, in some stateful workloads, careful design has been shown to offer tangible benefits (e.g., SplitServe [22]).
2) *Lambdas for transition:* When new VMs need to be added (the system is under-provisioned on purpose, or due to prediction errors, or as part of failure recovery), Lambdas, which have faster spin-up compared to VMs, can serve as transition mechanisms. Again, while this benefit is exploited for stateless workloads, it can also be used by stateful workloads through careful design [36].

B. FaaS to IaaS + FaaS

Currently, Lambdas are significantly expensive per-unit resources compared to VMs. This makes economic sense [21] – PaaS and SaaS are akin to "value-added services" in more conventional markets. Consider Amazon EC2 Instances and Amazon Lambdas as IaaS and FaaS, respectively. For 1 h, Lambda is approximately 3.5 times more expensive than a single vCPU on AWS *t3.medium* EC2 on-demand instance [3]. We configure Lambda memory to 4096MB as *t3.medium* has same memory size. [1]. Lambda's current per-GB-second invocation cost in us-east-1 is $0.0000166667, excluding Lambda's per-million-request pricing. With the *t3.medium* general-purpose instance type, one can obtain 4GB of memory and 2vCPUs for $0.0416/h.

Besides the costs stemming from these "primary" sources, a second contributor to the higher costs of PaaS comes from the shared-state management. Generally, PaaS-based products need to rely upon additional services to persist and share

[1] The amount of memory allotted to AWS Lambda is usually proportionate to the number of vCPUs, with one vCPU equating to approximately 1765MB.

Service Type	Billing Granularity	Spin-up Delays	Programming Difficulty	Price per unit resource
IaaS	Coarse	Very High	High	Very Low
PaaS	Fine	Low	High	High
FaaS	Very Fine	Low	High	High
SaaS	Very Fine	Very Low	Very Low	Very High

TABLE I: Comparison of different cloud service types.

state with associated costs. In particular, AWS Lambda-based stateful applications often rely upon S3 [6], Simple Queue Service (SQS) [8], shared caches (Memcached or Redis hosted on VMs or offered as PaaS/SaaS offerings), or shared file systems hosted on VMs (EFS) [11]. Similarly, Microsoft's Durable Functions [9] provide stateful orchestration of function execution. They are simply a collection of Azure Functions that use the Storage Account. The cost model for Azure Functions includes the number of executions, execution time, and storage consumption. The cost of Azure Storage does not begin to accrue until the first gigabyte is consumed, but if large amounts of data are stored in queues, the cost will increase substantially.

C. Cost Savings from Blending

Combining IaaS and FaaS has been found to save operational, as well as runtime, costs, and recent research has been performed on service blending using AWS Lambdas. Spock [20], SplitServe [22], and MArk [40] leverage FaaS as a transition mechanism while scaling-up IaaS resources and handling traffic spikes. Spock achieves cost savings in ML inference service queries by up to 33% compared to VM-based resource procurement schemes. SplitServe, an Apache Spark enhancement that integrates Amazon Lambdas and other cloud functions, reduces cloud expenses by up to 55% for workloads with minor-to-moderate amounts of shuffling and 31% for workloads with high amounts of shuffling, both when compared to VM-based auto-scaling. MArk, a cost-effective and SLO-aware ML serving system, has been demonstrated to obtain cost reductions of up to 7.8%, compared to the leading auto-scaling machine learning platform SageMaker.

D. Manual Overhead of Blending

Unfortunately, achieving optimal blending is far from trivial. From our experience, doing so necessitates extensive modifications to the application (usually covering both user-written and runtime/libraries), as well as substantial programmer time and effort. It is worth noting that, although *aaS and the emergence of "cloud-native" programming languages [10] make the construction of cloud-ready applications relatively easier, in most cases, they still require the application to be completely redesigned.

Consider, as an example, AWS Lambda, in which programmers typically spend time in converting original functions to Lambda-based functions, taking library modules, as well as global variables, into account. Program analysis is then necessary to determine function dependency, which results in the manual use of external storage for data dependency (AWS S3 [6] or EFS [11]); AWS Lambda is stateless, which implies that programming constraints exist when using storage for data flow. Finally, one must also construct a deployment zip file containing FaaS-ready code and use a cloud-provider tool to build the final Lambda code.

Due to the above-mentioned overhead, a tool for automating the creation of blending services, such as [16], [34], can be highly useful in practice. If customers are aware that transferring a specific section of code to FaaS will lower the cost, a means to automate the blending service using simple annotation schemes will provide flexibility and reduce requisite human work to rewrite the program. The primary objective of this study is to eliminate this manual effort by developing an automated compilation system that is capable of switching seamlessly between multiple blended versions based on user-specified requirements/constraints.

III. Overall Design of Splice

A. Design Space Exploration

While an automated compiler can alleviate the manual overhead of blending, several challenges exist in porting parts of the code to appropriate FaaS/IaaS services. Firstly, we need to develop an annotation schema [29] to perform code transformations. However, in order to develop an annotation scheme, two critical questions must be answered: (1) how would the compiler identify which function to offload?; and (2) how to design a systematic form of annotations for the compiler to recognize service selection; otherwise, users would have to edit code directly from the compiler's intermediate representation. It is worth noting that the inherent assumption here is that we consider tenants who have insights about application structure, resource needs, etc.

While addressing the challenges mentioned above, numerous related issues may arise even if there is a standard form for the annotations. In particular, developers may find that combining multiple sections per functional unit reduces wasteful network calls. Users might also provide certain limits, such as mapping some of the code to FaaS when a measure of interest (such as the number of requests over a given time or CPU/memory consumption) exceeds a specified threshold. Providing rules/constraints is critical because different resource offerings may benefit from different cloud settings, such as leveraging Lambda for parallelism and short-running jobs.

The requirements mentioned above necessitate the use of extra annotations for flexibility. Additionally, there is difficulty in communicating these rules/constraints to the load balancer, directing the servers to run blended services when acquiring additional resources. We need an intuitive way of integrating annotations with a load balancer. Although annotations can specify rules and constraints for blending services, challenges remain for blending, as they require extensive application changes, as discussed earlier in Sec. II-D.

B. Proposed Splice Design

Based on the discussion above, providing an automated means of integrating services can become crucial towards minimizing manual work. Towards this, we propose Splice,

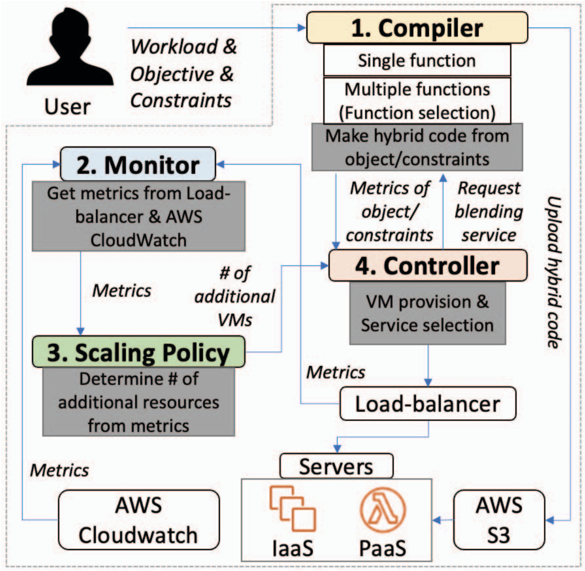


Fig. 1: High-level design of Splice depicting its various components.

which applies methodologies and tools to avoid the requisite programming and system processes to convert these insights into cloud-ready code. The high-level design specifications of Splice are shown in Fig. 1. Splice comprises a "pragma-based" annotation mechanism that the programmer may use to annotate the application code. The programmer's goal would be to convey domain knowledge of desirable "basic execution units" (BEUs), an execution granularity for blending selections that we would create. Splice systematically uses the annotations to identify specific BEU-to-service type mappings. Splice analyzes such annotated code and builds an intermediate representation (IR), which our proposed compiler employs to construct cloud-ready code automatically. The user specifies workload with annotations and an SLO.

We begin with the initial servers connected to the Load-balancer, which handles resource selection and procurement. The Splice engages when the Load-balancer decides to auto-scale. Splice interacts with its components and informs the Load-balancer to use blending services when applicable.

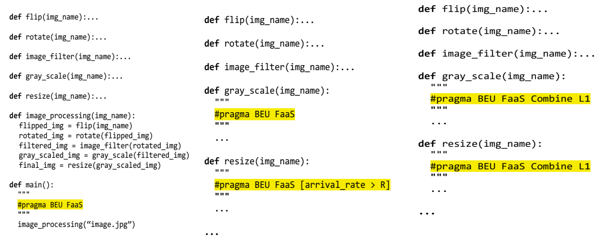
C. Component Details

*1) **Compiler**:* The Compiler is responsible for turning source code into *aaS executables. It begins by searching for annotated pragmas. If metrics are defined via user annotations, the Compiler forwards them to the Controller module. The Compiler then performs automatic transformation, enabling program analysis and automated creation of cost-effective cloud-ready programs. The Controller gathers information and orders the Compiler to upload executable code to all servers through AWS S3 once the Compiler has prepared the cloud-ready blended code.

*2) **Monitor**:* This module monitors AWS CloudWatch and Load-balancer. It collects CloudWatch (CPU and memory utilization of running servers) data and the Load-balancer's metric (the current request arrival rate). In the case of calculating the cost of Lambda, CloudWatch is also used for retrieving the billed duration of Lambda's execution. These measurements are conveyed to the Scaling Policy.

*3) **Scaling Policy**:* The Scaling Policy determines how many more instances will be made available to handle incoming requests. The Scaling Policy receives information about the current request arrival rate and metrics of servers from the Monitor module. For scaling-out resources, it calculates how many more VM instances are needed and sends the result to the Controller. For scaling-in resources, the Scaling Policy finds VMs that are idle for more than three minutes and sends them to the Controller; this heuristic is leveraged in order to avoid early termination of instances in the case of short-term request rate changes (as suggested by [19]). It is to be emphasized that Splice is capable of adapting to any scaling policy, including predictive and reactive scaling.

*4) **Controller**:* The Controller is responsible for allocating cloud resources and selecting services. It collects data from the Scaling Policy and user-annotated metrics from the Compiler. It is worth noting that, depending on how developers implement annotation, a blending service could imply merely using FaaS. The Controller can make several decisions based on the user-annotation type. Specifically for scaling-out resources, it can: (1) create more VM resources as a standard auto-scaling policy; (2) manage incoming requests utilizing blending services while generating more VM resources; or (3) employ blending services to handle incoming requests only when the current status meets the requirements of user-annotation metrics, such as executing on FaaS when the request rate exceeds the specified value. The Controller informs the Compiler to upload the blended cloud-ready code to all servers when the specified metric is satisfied. The Load-balancer later redirects requests to use the blended code to match incoming requests while acquiring additional resources.



```
def flip(img_name):...

def rotate(img_name):...

def image_filter(img_name):...

def gray_scale(img_name):...

def resize(img_name):...

def image_processing(img_name):
  flipped_img = flip(img_name)
  rotated_img = rotate(flipped_img)
  filtered_img = image_filter(rotated_img)
  gray_scaled_img = gray_scale(filtered_img)
  final_img = resize(gray_scaled_img)

def main():
  """
  #pragma BEU FaaS
  """
  image_processing("image.jpg")
```

(a) Offloading whole application

```
def flip(img_name):...

def rotate(img_name):...

def image_filter(img_name):...

def gray_scale(img_name):
  """
  #pragma BEU FaaS
  """
  ...

def resize(img_name):
  """
  #pragma BEU FaaS [arrival_rate > R]
  """
  ...
...
```

(b) Annotation for separate function

```
def flip(img_name):...

def rotate(img_name):...

def image_filter(img_name):...

def gray_scale(img_name):
  """
  #pragma BEU FaaS Combine L1
  """
  ...

def resize(img_name):
  """
  #pragma BEU FaaS Combine L1
  """
  ...
...
```

(c) Grouping functions to one Lambda function

Fig. 2: Illustrative image processing application snippet with annotations highlighted.

D. Annotation Schema

Recall that the term "Basic Execution Unit (BEU)" refers to a program fragment defined as a unit of entire application functionality. Explicit pragmas are applied to a collection of related functions considered appropriate for consideration as Lambda functions by developers (otherwise, the compiler sees a logic unit as VM). Our current compiler implementation

Pragma Type	Description
BEU FaaS	Placement on a Lambda function.
BEU FaaS [arrival_rate >R]	Placement on a Lambda function with load balancing rule.
BEU FaaS Combine L1	Grouping functions to the same Lambda function.

TABLE II: Pragmas supported in Splice.

identifies two types of pragmas, i.e., those representing a single function and those representing multi-functions.

To demonstrate how our proposed approach works in practice, let us consider the sample application code fragment in Fig. 2a, obtained from an image processing application that calls functions in sequential order. Each function affects the image generated by the previous function uniquely. A sample annotation of this fragment (with highlighted annotations) with several types of programmer-inserted BEUs for these functions is shown in Fig. 2.

The function *gray_scale*, from Fig. 2b, is designated as an appropriate function for insertion on a Lambda function as an example of a new type of annotation (pragma BEU FaaS). The example in Fig. 2b additionally shows how the user is leveraging domain knowledge to provide a formula for a Load-balancer's resource selection rule that Splice should consider (pragma BEU FaaS [arrival rate > R]), i.e., the compiler should insert a load balancing rule into the cloud-ready code that routes requests exceeding the specified threshold to the Lambda-based version of the function. The parameter R is later compared with the "current arrival rate" by Splice's Controller. The request rate is just one representative example that we use, while the Load-balancer rule is extendable to incorporate additional metrics with respect to the application constraints.

"BEU FaaS" in the main function declares that the entire program is a Lambda function, and the pragma BEU FaaS Combine L1 indicates that the compiler locates and unifies functions with the prefix "L1.", as illustrated in Fig. 2a and Fig. 2c, respectively. The ability to group one or more functions allows for some flexibility. Consider a group of BEUs with high point-to-point communication (state transfer) overhead. Combining BEUs to form a single execution unit (FaaS) may be the most cost-effective solution because it reduces network transfer delays on external storage, such as AWS S3, allowing FaaS to finish within its limited lifespan.

E. Application Representation

Splice parses the application code for programmer-inserted pragmas and generates an intermediate representation (IR). We see functions as logical units (BEUs) by default, apart from user-annotated BEUs. Table II lists the programmer-inserted annotations supported by Splice. Splice uses BEUs to create a graph form of the IR that expresses the service selection for each BEU, as well as relevant dependencies between them. Similar to a conventional call graph, each vertex/node in the graph represents a unique BEU, and the directed edge contains data exchange between BEUs. Using a graph data structure allows our compiler to automate the addition of the function's input/output to an external datastore (Amazon S3 [6]) and upload/download them using the corresponding external URL for data dependency between nodes.

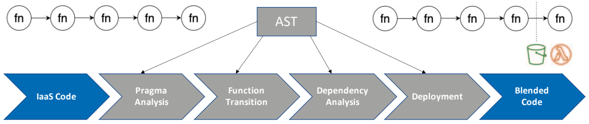


Fig. 3: Splice compiler pipeline.

IV. Implementation and Evaluation

In this section, we discuss in detail the implementation details of Splice on AWS using Python Abstract Syntax Tree (AST), followed by the evaluation methodology.

A. Implementation Details

Splice is implemented using the AST module for automatic code transformation.[2]

ASTs [32] are data structures that are commonly used in compilers to describe the structure of program code. They are essentially an intermediate representation of the program, which facilitates further analysis/transformations. We created a Splice compiler in Python with approximately 3,000 lines of code. The implementation pipeline for Splice is depicted in Fig. 3. It begins by detecting an annotated pragma. If the pragma contains metrics, it forwards them to the Controller, and then begins creating the corresponding Lambda code by examining the function name, parameters, libraries to pack, and global variables. It then searches for dependencies, such as function calls and data flow. Based on the dependencies, Splice automatically adds external storage (S3) and related library modules for the function's input and output. It takes data from the outcomes of dependencies and converts function calls to Lambda-invoked function calls. It finally generates the Lambda code via an AWS Command Line Interface using a deployment zip file that includes FaaS-ready code, library modules, and global variables.

B. Experimental Setup

Workload	Memory Allocated (MB)	Average Execution (ms)	Requests per vCPU
Resnet18 Inference	1712	129	6
Feature Generation	2048	358	2
Matrix Multiplication	2048	207	3
Image Processing	2048	6922	x

TABLE III: Workload description for VM and Lambda functions. The second column shows memory configuration for Lambda, and the fifth column represents the number of requests that are running in parallel per vCPU for VM.

To achieve a fair comparison, we change the memory configuration for Lambda (as shown in Table III) for an iso-performance experiment in which both VM and Lambda-based deployments execute in the same amount of time. Note that

[2] The current version of our implementation in Python is shared in https://github.com/mjaysonnn/Splice

we omit requests per vCPU for image processing application since it is used for a different scenario (see Sec. IV-B4). We limit our testing to instances from the same EC2 family (C5) to guarantee fairness which we chose because Lambdas have hardware comparable to C5.large VMs (2vCPUs, 4GB RAM) [37].

To handle millions of simultaneous requests, we use an AWS EC2 C5.2xlarge (8vCPUs, 16GB RAM) instance as a front-end with an Nginx load-balancer. Out of numerous load balancing methods (such as "Round-Robin", "IP Hash", or "Weighted Round-Robin"), we utilize the "Least Connection" method, which sends requests to the server with the fewest active connections to minimize idle VM resource consumption. AWS CloudWatch (a component of the Monitor) is used periodically in the front-end to collect CPU/memory utilization of running servers, the Load-balancer's metric, and billed duration of AWS Lambda. Note that the front-end's running time and the number of API requests of AWS CloudWatch are counted towards the VM and Lambda cost models.

To serve the individual requests, we use numerous C5.large instances. Each server comes with a library for executing workloads and FastAPI [17], a Python-based web platform that optimizes the server's resources. To avoid high data transfer costs, we conduct our tests in AWS's us-east-1 region.

To prevent cold starts on initial invocations, we manually invoke an idle Lambda function every 20min. Lambda functions are kept warm for at least 20 min after they are called [19], and the median cold start latency in AWS is less than 200ms. We take these extra invocations into consideration when calculating monetary costs. In addition to the cost of invocation and computation, we examine the cost of S3, which serves as stateful storage for retrieving the pre-trained model used in the inference workload and contains the deployment-code zip for creating Lambda functions.

1) Workload Generator: To assess the advantages of Splice, we developed a high-fidelity event-driven Workload Generator. The Request Generator (described further in Sec. IV-B2) provides input to the Workload Generator, which generates request arrival times based on real-world trace. The workloads used for evaluation are MXNet ResNet-18 model inference, feature generation, matrix multiplication, and image-processing applications (see Sec. IV-B3 and IV-B4).

2) Request Generator: We used WITS [39] and Wikipedia [38] (WIKI) traces as input to the Request generator. WIKI trace is a real-time record of users interacting with the Wikipedia website, with traffic patterns modeled by Poisson arrival times, short-term burstiness, and diurnal level fluctuations. Compared to the WIKI trace, the WITS trace contains a large variation in peaks. For the single function workloads (ResNet model inference, feature generation, matrix multiplication), we scaled both traces down to an average of 130 requests per second. For the multi-function workload (our image-processing application), we scaled the WITS trace down to an average of 65 requests per second due to its longer duration. This mimics a real-world situation in which the web service responds to different queries every second.

3) Single Function Workload Scenario: Each request from the Request Generator corresponds to running a separate single-function workload. We evaluate our proposed system using ML inference, feature generation, and matrix multiplication.

For inference queries, the ResNet-18 model (of size 120MB) is pre-trained using the Imagenet dataset, and we use the MXNet ML module [13] to deploy and perform model inference. We manually modified the serverless benchmark suites [24], created VM-based applications, and performed feature generation and image processing (cf. Sec. IV-B4). For feature generation, we use the Amazon Fine Food Review3 text dataset [4] as input. Each request for feature generation corresponds to fetching a review dataset from S3 and using the Pandas library [31] to transform it into a TF-IDF vector. For the matrix-multiplication application, we multiply two square matrices (each is of size 500 $\times$ 500) using the NumPy library [1].

Throughout the experiment, to manage 130 requests per second for both WITS and WIKI traces, we choose the number of initial servers depending on the number of requests each vCPU can handle in parallel per workload, subject to a response time goal of 1000ms (described in Table III). We use the first 60 minutes of the WITS and WIKI traces.

4) Multi-Function Workload Scenario: Each request from the Request Generator corresponds to execution of image processing application (shown in Fig. 2a). The application has five functions, i.e., *Flip*, *Rotate*, *Gray Scale*, *Filter*, and *Resize*, with response time of approximately 1.24s, 1.86s, 0.66s, 2.67s, and 0.3s, respectively. To run the application, we use C5.large VM instances and all Lambda functions have a RAM configuration of 2000MB for iso-performance experiments. The average response time for performing image processing is 7 seconds on a C5.large instance, and thus we use 455 ($= 7 \times 65$) servers to handle an average of 65 requests per second for the first 10 minutes of the WITS trace. Out of the 32 ($= 2^5$) ways of deploying sequential functions in VM or Lambdas or a blending, we sample 10 representative cases that exhibit variations in response time and monetary cost.

C. Evaluation Setup

We evaluate our results by comparing Splice's monetary cost and response time (as specified in the SLO) when deploying in the following resource acquisition scenarios: (1) *All Lambda*; (2) VMs with auto-scaling (*VM-autoscaling*); and (3) VM with conservative over-provisioning (*VM-overprovision*), i.e., 1.5 times the amount of resources needed. *VM-overprovision* is frequently employed in auto-scaling, to minimize SLO violations. To eliminate the effect of abnormal cloud noise, the workloads are performed five times.

The cost of VMs is calculated as a cumulative sum of the product of workload duration and the price per second of each VM. For the cost of Lambda functions, each Lambda function's pricing is determined by: 1) The number of times each function is invoked (N), 2) RAM allocated (M in GB), 3) execution time (E in sec) (specifically, billed duration stated in

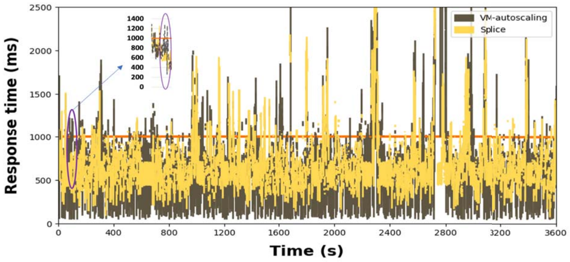


Fig. 4: Comparison of running ResNet-18 model inference between VM-only auto-scaling policy and Splice.

AWS CloudWatch), and 4) cost of 1 GB-s ($0.00001667/GB-s). Equation 1 captures the cost model (C) based on the above metrics (The constants below are modeled after Lambda function's pricing).

$$C = \big(M \times E \times 1.667 \times e^{-5} \times N\big) + \big(N \times 2.4 \times e^{-7}\big) \quad (1)$$

Besides the computation costs, storage costs are similar for VMs and Lambdas, and therefore they do not affect our cost comparisons. For the cost model of Splice and *All Lambda*, we also include the cost of the front-end server (EC2 C5.2xlarge instance) that uses Splice as well as AWS CloudWatch, which is used for retrieving metrics (CPU/memory usage) of servers and billed duration of Lambda functions.

The "response time" is defined as the end-to-end elapsed period between the time when the request is sent from the client and the time when the response is received from servers through the load-balancer (including communication to/from S3). "SLO violation" is calculated as the percentage of requests whose response time exceeds the given target (SLO).

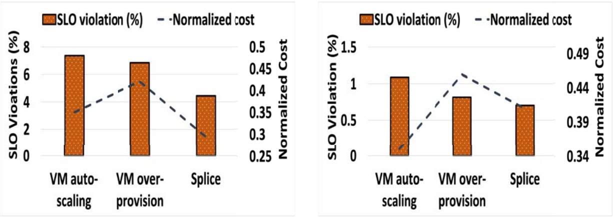
V. Analysis of Results

In this section, we present the major results from our evaluation of Splice in terms of SLO and cost, followed by sensitivity analysis, discussion, and future work from results.

A. *Benefits of Splice*

To demonstrate detailed advantages from Splice, Fig. 4 shows the comparison of ResNet-18 model inference workload produced by WITS between *VM-autoscaling* and Splice. The horizontal axis depicts the time, while the vertical axis represents the response time distribution per second, in a box-plot presentation with min and max, and an SLO of 1s. For readability, we omitted other important data such as percentiles and median response time.

When the requests spike and SLO violations occur (exceeding the orange line of the SLO), as shown in the purple circle of Figure 4, the scaling policy identifies and starts procuring new resources. For the conventional scaling policy (VM-autoscaling), however, requests still get queued on existing VMs. This is because new VMs are shown to have a start-up delay of 60s to 100s (the start-up delay is 61s and 64s, for VM-autoscaling and Splice, respectively), according to prior works [14], [27]. Due to VM's start-up delay, VMs stay active for long periods of time. Splice, on the other hand, automates the process of blending (making Lambda-based functions ready), and manages requests effectively via Lambda during resource acquisition, decreasing query queuing to current VMs and SLO violations. As a result, unused VMs created during the scale-out phase may be scaled in early.



(a) WITS

(b) WIKI

Fig. 5: SLO (1000ms) violations and cost reductions for ResNet-18 model inference under different scaling policies.

It is important to note that, whenever Splice decides to use blending, there is a compilation and data transfer overhead to inject the blended-code-snippet to the server. Note that Lambda functions have a size restriction of about 250MB for the deployment package when deployed from S3. This constraint will be discussed in Sec. V-F. For varying code snippet sizes ranging from 18MB to 230MB [3], we profile the time taken by Splice to compile and transfer the blended code from S3 to the servers. It is observed that this time ranges from 2s to 15s, which could constitute overhead for servers when employing blending. We also compute the blending overhead for feature generation, where Splice packs the code snippet, Panda library, and its dependent NumPy library (a total size of 100MB). The result shows that the overhead is about 10s. However, given that the new VMs start-up delays range from 60s to 100s, we overlap the blending overhead with the VM provisioning, which we believe is a reasonable technical design choice that we made in Splice.

B. *Single Function Scenario*

Fig. 5 demonstrates a comparison of SLO violation and cost savings for performing the MXNet ResNet-18 model inference service using WITS and WIKI under different scaling policies. The total number of requests is 469,394 and 463,599 for WITS and WIKI, respectively. The horizontal axis indicates different resource procurement schemes. The primary y-axis represents the percentage of SLO (1000ms) violations with response time. The secondary y-axis shows the monetary cost normalized to the *All Lambda*, resource procurement scheme that only uses Lambda.

It can be observed that, compared to the *VM-overprovision* policy, Splice lowers SLO violations by 16% and 13% for WITS and WIKI, respectively. The number of violated requests when using Splice is 62,733 and 13,371 for WITS and WIKI, respectively. The number of violated requests for *VM-overprovision* is 72,926 and 15,168 for WITS and WIKI,

[3] The code snippet size is primarily determined by the size of the parameters in the ML inference model.

	SLO violation (%)	Cost reduction (%)
SPLICE	0.134	32
SPOCK	0.129	34

TABLE IV: SLO violations and cost reduction of Splice and Spock for ResNet-18 model inference against *VM-autoscaling*.

respectively. The reduction of SLO violation is less than regular *VM-autoscaling* since more resources are available to serve requests during peak surges. However, because of these additional VMs, Splice significantly saves costs by 32% and 10% for WITS and WIKI, respectively.

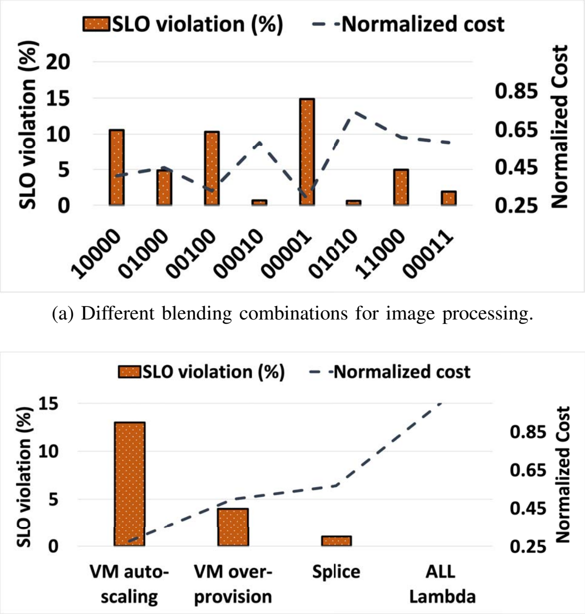
Splice also reduces SLO violations by 32% and 22% for WITS and WIKI, respectively, compared to *VM-autoscaling*. Specifically, one can see that the number of violated requests for *VM-autoscaling* is 82,625 and 16,348 for WITS and WIKI, respectively. At the same time, Splice lowers the cost of WITS by 6%. However, running the WIKI does not show cost savings. This is because, when compared to the WITS, the WIKI exhibits a less significant variance in peak-to-median request rates, resulting in few VMs being procured. Note that Lambda's per-unit price is higher than VM's per-unit price. Compared to less VM procurement, the cost of Lambda invocation leads to a negative effect on cost savings.

For the remaining workloads, feature generation and matrix multiplication, our proposed system shows a reduction in cost and SLO violations. We omit the graph since the results for single-function workloads are similar. Splice lowers SLO violations by up to 25% and 27% for feature generation and matrix multiplication. There is more reduction in SLO violations in these two workloads compared to the ResNet inference workload. This is primarily because these two workloads involve more CPU-intensive jobs where the fluctuations in request rate could lead to more resource contention, and Lambda could be more efficiently leveraged as a transition mechanism during provisioning (VM autoscaling) [20], [22]. Thus, Splice saves costs by up to 20% and 26%, respectively, for feature generation and matrix multiplication only for WITS. On the other hand, note that, using the WIKI results leads to no cost savings since, as previously stated, it has a lower variation in peak-to-median request rates.

C. Comparing the proposed method to Spock

Similar to Splice, Spock [20] exploits FaaS as a transition mechanism while handling load spikes when scaling-up new VMs. Spock converts single function into Lambda-based-ready function for blending in case of short-term request rate fluctuation. However, as stated in Sec. II, Spock incurs a manual overhead of creating blending services. To show the advantages of our proposal over Spock, we conduct an experiment comparing Splice and Spock in terms of SLO (response time target is 1000ms) and cost-reduction. Using the first 60 min of WITS trace with an average request rate of 130, we run workload of running MXNet inference over 5 iterations.

Table IV compares the cost saving and SLO violation reductions to the *VM-autoscaling*. The total number of requests is 469,394 , and the number of requests that violated SLO is 62,733 and 60,551 , for Splice and Spock, respectively. Both are measured to show benefits in terms of SLO violations and cost-reduction against *VM-autoscaling*. Although Splice shows similar benefits in ensuring SLO and monetary cost benefits, Splice alleviates the requisite work of blending.



(a) Different blending combinations for image processing.

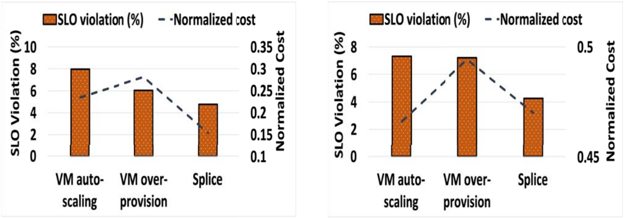
(b) Comparing the best blended version with VM and Lambdas for image processing.

Fig. 6: SLO violations and cost savings for image processing using WITS.

As mentioned above, compared to Spock, Splice *automates* manual overhead of turning code into FaaS-executable code. However, it faces additional overhead of compilation and data transfer at runtime. Based on our observation, this overhead of blending (turning MXNet ResNet-18 inference code to AWS Lambda code) while VM provisioning incurs only a marginal latency of 5.589ms. However, since the start-up duration of new VMs ranged from 61 to 94 seconds throughout the experiment, and Splice achieved comparable results to Spock, we believe Splice can outperform Spock in alleviating the requisite work of blending.

D. Multi-Function Scenario

Fig. 6a shows the SLO violation and cost savings for running the image processing workload using WITS under different blended versions. The workload comprises five functions, i.e., Flip, Rotate, Gray Scale, Filter, Resize, with each function's average response time described in IV-B4. Each function is executed in a VM when marked as 0 or in a Lambda when marked as 1. For example, "00010" means that only the fourth function, i.e., Filter function, is executed as a Lambda and the rest are executed in the VM. The total number of requests is 41,482. The primary y-axis indicates the percentage of SLO violations, and the secondary y-axis represents the total cost normalized to running the workload only on *All Lambda*.



(a) 500ms

(b) 1500ms

Fig. 7: SLO violations and cost savings for ResNet-18 model inference using WITS with different response time targets. 1000ms is shown in Fig. 5a.

We observe that “00010”(1%), “01010”(1%), and “00011”(2%) have the lowest SLO violations. These 3 combinations execute the most time-consuming function (*Filter* function) as Lambda. This allows the VMs to handle more requests for other functions, as they can be bin-packed more efficiently. However, offloading more functions (that have longer response time) to run as Lambda increases the cost. As a consequence, “01010” costs 28% more than “00010”. Therefore, the best-blended version for this particular trace is “00010”, i.e., when we execute only the Filter function as Lambda.

From the observations made above, Splice uses the best-blended version (“00010”) and we compare it against the *VM-autoscaling*, *VM-overprovision*, and *ALL Lambda* in Fig. 6b. The total number of requests is the same, i.e., 41,482 The primary y-axis indicates the percentage of SLO violations, and the secondary y-axis gives the normalized cost, as mentioned in the previous paragraph. We see that Splice reduces the SLO violations by 13% and 3% for *VM-autoscaling* and *VM-overprovision*, respectively, while costing 7% more than *VM-overprovision* case and 42% less than *All Lambda* case. This scaled WITS has a request rate of 65 or less for 47% of the entire duration, with a peak request rate of 135. If the peak request rate is increased, such that the workload is unpredictable, we can expect a greater reduction in cost and fewer SLO violations than any other policy. While the results that we present here are for one single application, we believe that the Splice framework has the capability to assess different function combinations for other applications and identify the most cost-efficient and SLO-aware solution.

E. Sensitivity Study

To demonstrate the efficiency of the proposed system with various response time requirements (SLOs), we evaluate Splice by running the MXNet ResNet-18 model inference service using WITS under 500ms, 1000ms, and 1500ms response time targets. To ensure fairness, we fix the number of total requests (469,394) and calculate the number of requests per vCPU for VMs for each given SLO. The number of requests per vCPU is 4, 6, and 9 for 500ms, 1000ms, and 1500ms, respectively. Note that we use the Workload Generator from WITS, whose average request rate is 130. For servers to handle enough requests throughout the experiment, we adjust the number of servers depending on the number of requests per vCPU. For Lambda functions, we adjust RAM configuration to meet different SLOs.

Fig. 7 shows a comparison of SLO violation and cost savings under different scaling policies on the severity of SLOs. The x-axis represents different resource procurement schemes, and the primary y-axis indicates the percentage of SLO violations. The secondary y-axis shows the normalized cost to the *ALL Lambda*. It is observed that Splice decreases the number of SLO violations compared to other scaling systems, irrespective of the severity of the SLO (up to 48%, 32%, and 23% for 500ms, 1000ms, and 1500ms, respectively). The numbers of violated requests for the standard *Splice* policy are 112,007 , 62,733 , and 60,238 for increasing order of the SLO. When the SLO increases, the maximum number of requests per vCPU for VMs increases, withstanding the sudden rise in latency (The number of instances that were created and terminated during the experiment is 64 and 24 for 500ms and 1500ms, respectively.) As a result, the SLO violation gap between *VM-autoscaling* and *VM-overprovision* narrows. However, since Lambda is serverless and used to handle millions of requests, Splice is unaffected by the limit on requests per vCPU.

When compared to the *VM-overprovision* method and running workloads only on Lambda, Splice shows cost benefits regardless of SLO. However, compared to the *VM-autoscaling* approach, cost savings are inconsistent. As the execution-time target increases (SLO slackens), Lambda’s RAM configuration value decreases, but Lambda’s execution time increases (due to the iso-performance comparison). Considering the cost model of VM and Lambda, increasing SLO results in a much higher FaaS cost. The cost savings of Splice compared to *VM-autoscaling* policy declines and eventually vanishes as the execution-time target increases (34%, 16%, -1%). This demonstrates how Lambda settings (RAM/number of invocations) and latency-sensitivity may influence the benefits brought by Splice.

F. Discussion and Future Work

In this subsection, a few design limitations of Splice are identified, along with our planned future work. Firstly, we plan to reduce the overhead of compiling and transferring the blended code from S3 to the servers while blending. Instead of sending the entire code-snippet to the server, we plan to use the de-duplication technique to transfer only the modified portions of the code. Secondly, Splice creates Lambda code by downloading a zip package from S3, which could be quite time-consuming. As an alternative, we plan to experiment with the AWS Elastic File System (EFS) to provide shared file access [11] to all Lambdas. Subsequently, we plan to conduct an extensive study to determine optimal storage solution based on workload and user requirements.

From our observations with respect to the higher cost of using Lambdas for a relaxed SLO (Sec. V-E), we plan to also include execution time as a metric in Splice. Based on available slack in SLO, Splice can tune Lambda configurations such that cost is minimized. In addition, Splice's automatic code transformation is dependent on programming framework characteristics, such as libraries, attributes, and class methods. We will bypass these limits by using additional optimization, such as Lambda lifting. We plan to evaluate Splice for large-scale applications containing 10s to 100s of functions, and also investigate blending other cloud services, such as SaaS.

VI. Discussion of Related Work

Compiler Optimizations: Compiler optimizations have been extensively incorporated into numerous domains through certain techniques, such as annotation schemas [28]–[30], [35] and source-to-source translators [12]. Weld [28] and LLVM [26] aim to demonstrate the benefits offered by common intermediate representations (IRs) for a variety of applications. In contrast, Splice combines annotations and compiler optimizations to generate *blended cloud-ready* code. In the cloud domain, few works have focused on compiler-driven code transformation. "FaaSification" [34] produces modules that dynamically decompose, convert, and deploy unmodified code as AWS Lambda functions. Similarly, "Costless" [15] automatically transforms code written with cloud functions to take advantage of cost savings from function fusion and splitting. We argue that our method is quite flexible, as it can shift the specific portions of the code to FaaS instead of the entire workload, which is deemed expensive, especially for long-running workloads.

Service Blending: Blending resources, especially IaaS with FaaS, has been thoroughly discussed in Sec. II. Blending within IaaS has been extensively used in the past where different-sized VMs, transient VMs or etc. [36], are multiplexed for cost and performance efficiency. In terms of FaaS, highly parallelizable applications, such as video analytics, have been entirely redesigned to Lambda functions to achieve lower latencies at reduced cost [18], [23].

VII. Concluding Remarks

In this work, we demonstrate the effectiveness of combining several types of IaaS with FaaS, and present an automated way of combining various resource offerings. We propose Splice, an automated cost- and performance-aware platform for blending IaaS and FaaS services based on users' modest pragma annotations of the source code. Our detailed experimental evaluation using various workloads indicates that Splice minimizes SLO violations of ML inference queries by up to 31%, while saving up to 32% on costs, compared to the standard resource procurement methods.

Acknowledgement

This material is based upon work supported by the National Science Foundation under grant numbers 2122155, 2028929, 2008398, 1931531, 2119236, and 1908793.

References

[1] Array programming with NumPy. *Nature*, 585:357–362, 2020.

[2] O. Alipourfard et al. Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *Proc. NSDI*, 2017.

[3] ec2-pricing. http://aws.amazon.com/ec2/pricing.

[4] Amazon fine foods reviews. https://snap.stanford.edu/data/web-FineFoods.html.

[5] AWS Elastic Container Service. https://aws.amazon.com/ecs/.

[6] AWS S3. https://aws.amazon.com/s3/.

[7] AWS Sagemaker. https://docs.aws.amazon.com/sagemaker/.

[8] Amazon sqs. https://aws.amazon.com/sqs/.

[9] Microsoft Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[10] A. Balalaie et al. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 2016.

[11] James Beswick. Using Amazon EFS for AWS Lambda in your serverless applications. https://aws.amazon.com/blogs/compute/using-amazon-efs-for-aws-lambda-in-your-serverless-applications/.

[12] Christian Champagne. Method for transforming first code instructions in a first programming language into second code instructions in a second programming language, US Patent 10,324,695 B2, June 2019.

[13] T. Chen et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. https://arxiv.org/abs/1512.01274.

[14] A. Chung, J. Park, and Gregory R Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proc. ACM SOCC*, 2018.

[15] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In *Proc. IEEE/ACM Symp. on Edge Computing*, 2018.

[16] FaaSification Tools to Help You Create Functions-as-a-Service Faster. http://www.faasification.com/.

[17] Fastapi framework. https://fastapi.tiangolo.com/.

[18] S. Fouladi et al. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proc. NSDI*, 2017.

[19] A. Gandhi et al. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 2012.

[20] J.R. Gunasekaran et al. Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud. In *IEEE CLOUD*, 2019.

[21] D. Irwin and B. Urgaonkar. Nsf workshop on cloud economics, 2018. https://umass-sustainablecomputinglab.github.io/nsfw/.

[22] A. Jain et al. SplitServe: Efficient Splitting Complex Workloads across FaaS and IaaS. In *Proc. ACM/IFIP Middleware*, 2020.

[23] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. ACM SOCC*, 2017.

[24] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *IEEE CLOUD 2019*.

[25] AWS Lambda. https://aws.amazon.com/lambda/.

[26] The LLVM Compiler Infrastructure. https://llvm.org/.

[27] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *IEEE CLOUD*, pages 423–430. IEEE, 2012.

[28] S. Palkar et al. Weld: A common runtime for high performance data analytics. In *Proc. CIDR*, 2017.

[29] S. Palkar and M. Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *Proc. SOSP*, 2019.

[30] Shoumik Palkar and Matei Zaharia. Splitability annotations: Optimizing black-box function composition in existing libraries. *CoRR*, 2018.

[31] The pandas library. pandas-dev/pandas: Pandas, February 2020.

[32] python ast module. https://docs.python.org/3.6/library/ast.html.

[33] V.M. Do Rosario et al. Fast and low-cost search for efficient cloud configurations for hpc workloads. https://arxiv.org/abs/2006.15481, 2020.

[34] J. Spillner, C. Mateos, and D. A. Monge. FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC. *CARLA*, 2017.

[35] A. K. Sujeeth et al. OptiML: An Implicitly Parallel Domain-specific Language for Machine Learning. In *Proc. ICML*, 2011.

[36] C. Wang et al. Combining spot and on-demand instances for cost effective caching. In *Proc. EuroSys*, 2017.

[37] L. Wang et al. Peeking behind the curtains of serverless platforms. In *Proc. USENIX ATC*, 2018.

[38] Wikimedia database dump. https://archive.org/details/enwiki-20180320.

[39] WITS: Waikato Internet Traffic Storage. https://wand.net.nz/wits/.

[40] C. Zhang et al. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proc. ATC*, 2019.